JNI 함수 호출을 통한 C에서의 자바 객체 사용1)

이창환⁰ 오세만
동국대학교 컴퓨터공학과
{yich⁰, smoh}@dongguk.edu

Using Java Objects in C through the JNI Function Calls

Chang-Hwan Yi⁰ Se-Man Oh
Dept. of Computer Engineering, Dongguk Univ.

요 약

JNI는 자바와 네이티브 코드간에 상호 연동을 위해서 사용되는 인터페이스이고, JNI를 이용하면 C에서 자바 객체를 사용할 수 있다. C에서 자바 객체에 대한 연산을 하기 위해서는 객체 연산의 종류에 따른 일정한 JNI 함수 호출 패턴을 이용해야 한다. 사용자가 직접 자바에 대한 연산을 기술하는 경우, 사용자는 복잡한 함수 호출 패턴을 익히고 패턴에 필요한 정보를 직접 입력해야 하며, 패턴의 잘못된 기술과 올바르지 않은 정보의 입력에 따른 오류 발생할 가능성이 높은 문제점이 있다.

본 논문에서는 자바에서 점(".") 연산자를 사용하여 객체에 대해 연산하는 것처럼 C에서도 점 연산자를 사용하여 자바 객체에 대한 연산할 수 있는 방법을 제안하고 구현하였다. 제안된 방법은 점 연산자를 사용한 자바 객체에 대한 연산을 같은 의미를 가지는 여러 JNI 함수 호출로 변환하는 것으로, 사용자가 직접 기술해서 발생하는 여러 문제점을 제거하여 사용의 복잡성과 오류 생성의 발생 가능성을 줄이는 장점을 가지고 있다.

1. 서론

자바는 자바 실행 환경(Java Runtime Environment)이 아닌 다른 환경에서 자바 객체를 사용할 수 있도록 JNI(Java Native Interface)라는 방법을 제공하고 있다.[1] JNI를 사용하면 C에서 자바 객체의 필드 값을 읽어오거나 변경하고, 메소드 호출하는 것과 같은 자바 객체에 대한 연산을 할 수 있다.

C에서 자바 객체에 대한 연산을 하기 위해서는 객체 연산의 종류에 따른 일정한 JNI 함수 호출 패턴을 이용해야 한다. 사용자가 직접 자바에 대한 연산을 기술하는 경우, 사용자는 복잡한 함수 호출 패턴을 익히고 패턴에 필요한 정보를 직접 입력해야 하고, 패턴의 잘못된 기술과 올바르지 않은 정보의 입력에 따른 오류 발생할 가능성이 높은 문제점이 있다.

본 논문에서는 자바에서 점(".") 연산자를 사용하여 객체에 대해 연산하는 것처럼 C에서도 점 연산자를 사용하여 자바 객체에 대한 연산할 수 있는 방법을 제안하고 구현할 것이다.

2. 배경연구

2.1 자바와 C간의 변환기

현재까지의 자바와 C간의 변환에 대한 연구는 자바 실행 성능을 향상시키기 위해 자바 소스를 같은 의미의 C나 C++ 소스로 변환하는 것이었다. 그리고 이전 C 소스를 재활용하기 위해 C 소스를 자바 바이트 코드로 컴파일하여 자바 가상 기계에서 실행할 수 있도록 연구도 있다.[2][3]

그러나 본 논문에서는 성능 향상을 위한 자바와 C 사이의 변환이 아닌, JNI의 사용 편이성을 개선하기 위한 방법으로 C에서 자바 객체를 사용할 수 있는 방법을 제안하려 한다.

1) 본 연구는 한국과학재단의 특정기초연구(과제번호:1999-1-30300-3) 지원에 의한 것임.

2.2 자바 객체 연산에 따른 JNI 함수 호출 형태

C로 구현되는 네이티브 메소드에서는 JNI를 사용하여 객체의 필드 값을 읽어오거나 변경하고, 메소드를 호출하는 것과 같은 자바 객체에 대한 연산을 할 수 있다.

JNI를 사용한 자바 객체에 대한 연산을 하기 위해서는 연산할 자바 객체의 멤버에 대한 참조와 연산을 수행하는 JNI 함수, 이 JNI 함수의 인자에 대한 정보가 필요하다. 멤버에 대한 참조를 구하기 위해서는 클래스에 대한 참조와 멤버 이름, 멤버의 시그네처가 필요하고, 클래스에 대한 참조는 네이티브 메소드의 인자에서 구하던지 객체에 대한 참조에서 *GetObjectClass*() 함수를 통하여 구할 수 있다.

각 자바 객체의 연산을 수행할 때, 필요한 JNI 함수 호출 패턴을 정리하면 [그림 1], [그림 2]와 같다.[4][5]

```
/* jobject obj: 연산할 자바 객체 참조 */
jclass clazz = (*env)->GetObjectClass(env, obj);
jfieldID fid = (*env)->GetFieldID(env, clazz,
                                  "<name>",
                                  "<type sig>");

// Read Instance Field Value
val = (*env)->Get<type>Field(env, obj, fid);

// Read Class Field Value
val = (*env)->GetStatic<type>Field(env,
                                   clazz, fid);

// Write Instance Field Value
(*env)->Set<type>Field(env, obj, fid, val);

// Write Class Field Value
(*env)->Set<type>StaticField(env,
                             clazz, fid, val);
```

그림 1. 자바 객체의 필드 값 읽기와 변경하기

```
/* jobject obj: 연산할 자바 객체 참조 */
jclass clazz = (*env)->GetObjectClass(env, obj);
jmethodid mid = GetMethodID(env, obj,
        "<MethodName>", "<Signature>");

// Invoke Virtual Method
(*env)->Call<return_type>Method(env, obj, mid,
                                  ...);

// Invoke Non-Virtual Method
(*env)->CallNonVitural<return_type>Method(env,
                        obj, mid, ...);

// Invoke Static Method
(*env)->CallStatic<return_type>Method(env, clazz,
                                  mid, ...);
```

그림 2. 자바 객체의 메소드 호출하기

2.3 자바 객체 연산에 필요한 정보

네이티브 메소드에서 자바 객체에 대한 연산을 자동으로 JNI 함수 패턴으로 변환하기 위해서는 다음과 같은 정보들이 필요하다.

- 멤버 이름
- 멤버가 속한 클래스
- 멤버가 필드, 메소드 여부
- 멤버의 시그네춰(signature)
- 멤버가 인스턴스 멤버, 클래스 멤버 여부
- 멤버가 메소드인 경우, 버추얼 메소드, 논-버추얼 메소드 여부

위 정보에서 멤버가 속한 클래스에 대한 정보는 사용자에 의해서 제공이 되어야 한다. 나머지 정보는 자바의 리플렉션(reflection) API를 통해 자바 클래스 라이브러리에서 읽거나, C 소스 코드를 분석해서 추출해야 한다.

3. 설계

3.1 시스템 구조

자바 객체에 대한 연산을 JNI 함수 패턴으로 변환하는 시스템의 구조는 [그림 3]과 같다. 이 시스템은 점 연산자를 사용한 자바 객체 연산을 기술한 *.jc 파일을 입력으로 받아, 객체 연산을 JNI 함수 호출로 변환하여 *.c의 C 소스로 출력한다.

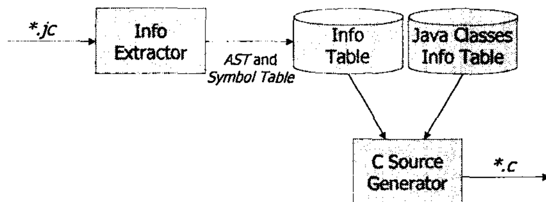

그림 3. 시스템 구조도

3.2 구현 환경 및 실행 환경

구현 환경은 다음과 같다.

- 한글 Windows 2000 Professional
- Java Development Kit 1.3.1_02

실행 환경은 다음과 같다.

- Java2 Runtime Environment

3.3 입력 파일 형식

입력 파일은 자바 객체 연산과 자바 객체 연산에 필요한 정보를 기술한 파일이다.

자바 객체에 대한 연산은 JNI 네이티브 메소드를 구현한 함수안에 점 연산자를 사용하여 나타내고 있다. 자바 객체 연산를 위해 추가되는 정보는 입력 파일에서 사용하는 패키지에 대한 정보, 네이티브 메소드가 속한 클래스 이름, 메소드 이름, 메소드 시그네춰등이다. 패키지에 대한 정보는 *#import <"패키지 이름">*과 같은 형태로 클래스보다 앞에 선언한다. 네이티브 메소드에 대한 정보는 자바 문서화 도구인 javadoc에서 사용하는 형태를 따르고 있다.

```
#import "<package_name>"

...

/**
 @class        class_name
 @method       method_name
 @signature    signature
 @param        ...
 */
JNIEXPORT <return_type> JNICALL
Java_<class_name>_<method_name>(JNIEnv* env, ...)
{
    ... // Java Object Operations in C
}
```

그림 4. 입력 파일 형식

4. 구현

4.1 정보 추출기

정보 추출기는 입력 파일에서 자바 객체 연산을 변환하고, C 소스를 생성할 때 사용되는 정보를 추출한다. 정보 추출기는 기본적으로는 C 파서이지만, 입력 파일에 새로 추가된 정보를 처리 할 수 있도록 파서를 수정하였다.

파서는 C 파서에 대한 입력 예제 파일을 가지고 있어, 쉽게 C 파서를 생성할 수 있는 JavaCC 파서 생성기를 사용하였고, 정보 추출기는 JavaCC 입력 파일을 수정하여 구현하였다.[6]

수정된 부분은 다시 C 소스로 생성할 수 있도록 입력 파일 정보를 정보 테이블에 저장할 수 있도록 변경하였고, 네이티브 메소드에 대한 정보를 수집하여 정보 테이블에 추가하도록 하였다. 또한 입력 파일 안에서 사용되는 자바 패키지에 대한 정보도 수집할 수 있도록 하였다. 그리고 각 환경마다 추가적으로 사용되는 키워드를 처리할 수 있도록 하였다.

4.2 정보 테이블

정보 테이블은 소스 파일의 AST(Abstract Syntax Tree) 정보, 심볼 정보, 결과 C 소스 생성을 위한 원 C 소스 정보, 네이티브 메소드 정보, 사용 자바 패키지 정보등을 가지고 있다.

이 중 AST와 심볼 정보는 파서를 통해 일반적으로 생성되는 정보이지만, 나머지 정보는 변환을 위해서 사용이 되는 정보이다. 이 추가된 정보는 AST의 노드 정보를 확장하여 추가된다.

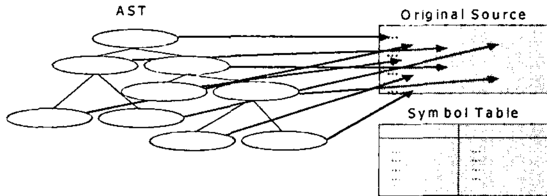

그림 5. 정보 테이블 구조

4.3 C 소스 생성기

C소스 생성기는 정보 테이블에 저장된 C 소스의 AST 검색하여 네이티브 메소드내의 자바 객체 연산을 찾아, 이를 해당하는 일련의 JNI 함수 호출로 변환한다.

생성기는 C 소스를 생성하기 위해서 AST를 탐색하여 자바와 관련이 없는 코드는 C 소스에 그대로 출력한다. 자바와 관련된 코드는 네이티브 메소드인 경우에는 변환을 위해서 노드를 더 탐색하고, 생성기에 정보를 제공하기 위한 노드의 경우에는 정보를 얻고, 노드를 무시한다. 생성기가 네이티브 메소드를 검색하는 경우, 객체에 대한 연산을 찾기 위해서, 점 연산자가 사용된 수식을 찾는다. 다른 노드의 경우에는 그대로 결과 파일에 출력한다. 점 연산자가 사용된 수식을 찾으면, 연산자의 피연산자가 자바 객체인지 C 구조체인지를 판별한다. 만약 자바 객체라면, 점에 사용된 자바 객체 연산이 필드에 대한 연산인지, 메소드에 대한 연산인지를 구분해야 한다. 필드에 대한 연산이라면, 필드 값을 읽는 것인지 값을 변경하는 것인지를 구분해야 한다. 위와 같은 구분 과정을 거쳐, 사용된 자바 객체 연산의 종류에 대한 정보와 정보 테이블에 저장된 객체에 필요한 정보를 사용하여 코드 생성에 사용할 코드 형태를 찾아내어 결과 파일에 변환된 코드를 출력한다.

5. 실험결과

본 논문에서는 [그림 5]와 같은 자바 객체의 연산을 가진 C 소스를 변환하여 [그림 6]과 같은 결과를 얻었다.

[그림 5]의 입력은 JNI를 사용하여 네이티브 메소드를 구현한 것으로, 메소드내에서 자바 객체의 두 필드 값을 읽고 메소드를 호출하는 것을 나타낸 것이다. 이 입력을 논문에서 제안된 방법으로 변환하면, 점 연산자를 사용한 자바 객체 연산이 해당하는 일련의 JNI 함수 호출로 변환된 [그림 6]과 같은 결과를 얻을 수 있다.

```
JEXPORT void JNICALL
Java_Exam_NativeMethod(JNIEnv *env, jobject obj)
{
    jint a, b;
    a = obj.FieldA;
    b = obj.FieldB;
    obj.MethodA();
}
```

그림 6. 입력 소스

```
JEXPORT void JNICALL
Java_Exam_NativeMethod(JNIEnv *env, jobject obj)
{
    jclass clz;    //
    jfieldID fid;    //
    jmethodID mid;   //
    jint a, b;

    /* a = obj.FieldA;*/
    clz = (*env)->GetObjectClass(env, obj);
    fid = (*env)->GetFieldID(env,
                            clz, "FieldA", "I");
    a = (*env)->GetIntField(env, obj, fid);

    /* b = obj.FieldB;*/
    clz = (*env)->GetObjectClass(env, obj);
    fid = (*env)->GetFieldID(env,
                            clz, "FieldB", "I");
    b = (*env)->GetIntField(env, obj, fid);

    /* obj.MethodA(); */
    clz = (*env)->GetObjectClass(env, obj);
    mid = (*env)->GetMethodID(env, clz,
                            "MethodA", "()V");
    (*env)->CallVoidMethod(env, obj, mid);
}
```

그림 7. 변환 결과

6. 결론 및 향후 연구

본 논문에서는 C에서 점 연산자를 사용한 자바 객체 연산을 같은 의미의 JNI 함수 호출 패턴으로 변환하는 방법을 제안하고 구현하였다. 제안된 방법을 사용하면 C에서의 자바 객체 연산의 복잡성과 사용자에 의해 발생할 수 있는 오류의 가능성을 줄이는 장점을 가지고 있다.

향후 연구로는 C에서 "+" 연산자와 "[]" 연산자를 사용한 자바 문자열 객체와 배열 객체에 대한 연산을 같은 의미의 JNI 함수 호출 패턴으로 변환하는 방법을 연구할 것이다. 또한 자바 객체에 대한 연산을 C++에서도 변환 가능도록 하고, C++의 new 연산자를 통해 자바 객체의 생성과 생성된 객체의 참조를 반환받고, delete 연산자를 통해 객체 참조를 제거할 수 있는 방법도 연구할 것이다. 그리고 객체에 대한 연산을 해당 패턴으로 1:1 변환하면, 결과에 필요 없는 JNI 함수 호출이 나타나고, 필요 없는 함수 호출을 줄여 JNI 실행 성능을 향상시키는 방법에 대한 연구도 필요할 것이다. 다음으로 자바 코드와 네이티브 코드를 같은 파일에서 기술할 수 있는 JNI 전처리기인 JPP에 본 논문에서 제안 방법을 통합하여 자바 네이티브 메소드를 쉽게 구현할 수 있는 일관된 방법을 제공하도록 할 것이다.[7]

참고문헌

[1] Java Native Interface, Javasoft, 1997.
[2] Toba: A Java-to-C Translator, http://www.cs.arizona.edu/sumatra/toba/
[3] GCJ: The GNU Compiler for Java, http://gcc.gnu.org/java/
[4] Compione, Walrath, Huml, and Tutorial Team, "Java Tutorial continued", Addison-Wesley, 1998.
[5] Rob Gordon, "Essential JNI", Prentice-Hall, 1998.
[6] JavaCC, http://www.webgain.com/products/java_cc/
[7] 이창환, 오세만, "JPP: JNI 전처리기", 정보처리학회 논문지 9-A권 1호